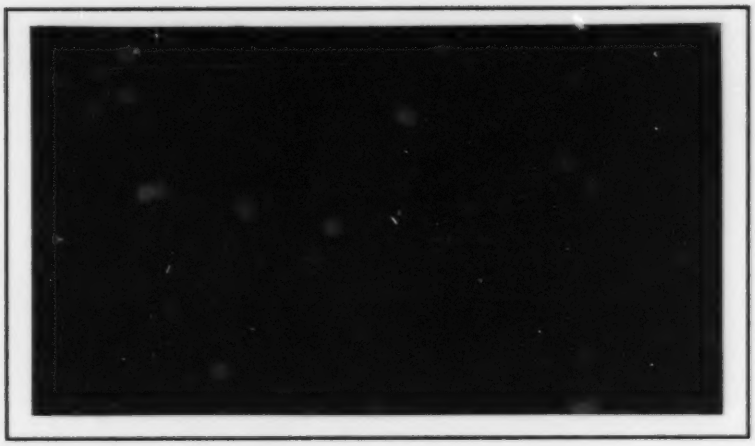
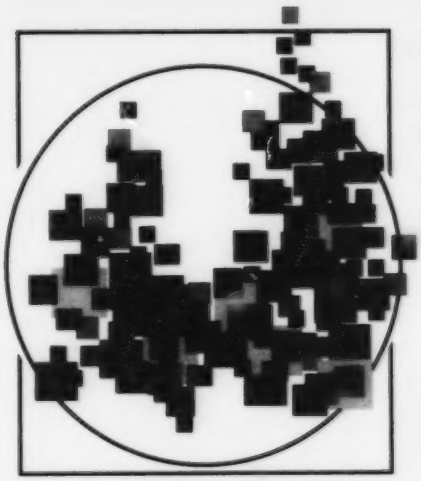




CENTRE FOR
SYSTEMS
SCIENCE

LCCR

Simon Fraser University, Burnaby, B.C., Canada V5A 1S6 / (604) 291-4704

# Compilation and Evaluation of Linear Mutual Recursions

Jiawei Han†

CSS/LCCR TR 90-06

April 1990

92-05790/

# Compilation and Evaluation of Linear Mutual Recursions

**Jiawei Han** †
*School of Computing Science*
*Simon Fraser University*
*Burnaby, British Columbia, CANADA V5A 1S6*
*han@cs.sfu.ca*

**Abstract.** *A linear mutual recursion is a recursion which consists of a set of rules, each of which has at most one predicate in its body which is mutually recursive with its head predicate. This paper studies the compilation and efficient evaluation of linear mutual recursions. It demonstrates that a large class of linear mutual recursions can be compiled to one or a set of linear recursions or pseudo-linear recursions and evaluated efficiently using the well-studied linear recursive query processing techniques and/or their variations. Moreover, it shows that the size of the compiled pseudo-linear recursion for a complex linear mutual recursion may grow exponentially in the worst case. The techniques for compilation and evaluation of linear mutual recursions are studied in the paper.*

**Keywords**. *Deductive databases, compilation techniques, linear recursions, mutual recursions*, and *recursive query evaluation.*

## 1. Introduction

Mutual recursions occur frequently in logic programming and deductive databases programs. Moreover, the rule rewriting techniques of the bottom-up evaluation methods [1-3, 12] may also generate mutually recursive rules. Therefore, it is necessary to study the efficient evaluation of mutual recursions.

There have not been many studies on the special compilation and evaluation techniques for mutual recursions. Most studies treat mutual recursion as one of the most complex kinds of recursions and suggest its evaluation by the general recursive query processing techniques, such as the magic sets method [1], the generalized magic sets method [3], the query/sub-query approach [13], or the PROLOG implementations [11]. Zhuang and Henschen [14] classified mutual recursions and studied the compilation of some classes of mutual recursions. Unfortunately, their compilation of many classes of mutual recursions generates quite complex formulas which can hardly be evaluated

† The work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant A-3723 and a research grant from Centre for System Science of Simon Fraser University.

efficiently.

In this paper, we study the compilation of a popular class of mutual recursions, *linear mutual recursions*, in deductive databases. A linear mutual recursion is a recursion which consists of a set of rules, each of which has at most one predicate in its body which is mutually recursive with its head predicate. Our study shows that a large class of linear mutual recursions can be compiled to one or a set of linear recursions or pseudo-linear recursions and evaluated efficiently using the well-studied linear recursive query processing methods [1, 2, 7, 8, 12] and/or their variations. The techniques for compilation and evaluation of linear mutual recursions are developed in the paper. Moreover, we point out that the size of the compiled recursion for a complex linear mutual recursion may grow exponentially in the worst case.

The paper is organized as follows. Section 2 introduces the concepts of IDB dependency graph and linear mutual recursion. Section 3 studies the compilation of linear mutual recursions. Section 4 examines the efficient evaluation of the compiled linear mutual recursions. Section 5 discusses the strength and the limitations of the compilation and evaluation methods, and section 6 summarizes our discussion.

## 2. IDB Dependency Graphs and Linear Mutual Recursions

Like many researchers, we assume that a deductive database (DDB) consists of an *extensional database (EDB)* (a set of extensional facts) and an *intensional database (IDB)* (a set of intensional predicates defined by *function-free* Horn-clauses). Without losing generality, we assume that there is no overlap between IDB and EDB predicates [12]. As a notational convention, we adopt a PROLOG-like notation except using the upper case letters for predicates and the lower case letters for constants and variables.

**Definitions.** A predicate $P$ is said to **imply** a predicate $R$ ($P \Rightarrow R$) if there is a Horn clause in *IDB* with predicate $R$ as the head and $P$ in the body, or there is a predicate $Q$ where $P \Rightarrow Q$ and $Q \Rightarrow R$ (transitivity). A predicate $R$ is **recursive** if $R \Rightarrow R$. A set of predicates are **mutually recursive** if for any two predicates $R$ and $S$ in the set, $R \Rightarrow S$, and $S \Rightarrow R$. A set of mutually recursive predicates form a **clique**.

To facilitate the compilation and processing of a deductive database, we associate with each predicate a **deduction level**. Each EDB predicate is at level 0. An IDB predicate is at least one level above the maximum level of all of its nonmutually recursive predicates in the bodies of the rules defining it and is at the same level as its mutually recursive predicates. Such a deduction level assignment ensures that each IDB predicate can be computed using the lower-level predicates plus the computation of a recursion, if any. In this way, an IDB is stratified by deduction levels. Notice that such a stratification is based on the order of the computation of IDB predicates, which is different from the popularly discussed *stratified logic programs* [12] in which the stratification of a database is based on the notion of negation.

The algorithm below computes the deduction level for each predicate in a deductive database.

**Algorithm 1.** Computing the deduction levels for each predicate in a deductive database.

**Input.** A set of Horn-clause rules.

**Output.** The deduction level of each predicate in the rule set.

**Method.**

FOR each predicate $P$ in the rule set DO level[$P$] := 0;
REPEAT
  FOR each rule $r$ with the head predicate $P$ DO
    For each predicate $R$ in the body of rule $r$ DO
      IF level[$R$] ≥ level[$P$] and $R$ is not mutually recursive with $P$
      THEN level[$P$] := level[$R$] + 1
      ELSE IF level[$R$] > level[$P$] and $R$ is mutually recursive with $P$
      THEN level[$P$] := level[$R$];
UNTIL there is no change. □

**Theorem 1.** The algorithm 1 terminates and correctly computes the deduction level for each predicate in a deductive database.

Proof sketch.

If $P$ (the head predicate) and $R$ (a predicate in the body) are mutually recursive predicates, $P$ is assigned to the same level as $R$ at the second assignment statement in the loop. Otherwise, $P$ will be at a higher level than $R$ after the first assignment statement in the loop. Therefore, the assignments derive correct deduction levels if the algorithm terminates. Since only non-mutually recursive predicates may pass the condition of the first assignment statement, such an assignment cannot be circular. Thus the algorithm terminates. □

To facilitate our analysis, we draw an *IDB dependency graph* for each IDB.

**Definition.** An **IDB dependency graph** is a directed graph defined as follows. For each IDB predicate $R$, there is a vertex $R$ in the graph. For the set of rules with head predicate $R$ and defined solely by EDB predicates, we draw a small circle surrounding $R$ ($R$ is called a *circled node*). For each IDB predicate $S$ in the body of a rule with the head predicate $R$, there is a directed edge $S \rightarrow R$. An edge or a circled node can be labeled by its corresponding rule identifier(s) when necessary.

The deduction level of each IDB predicate can also be determined using the IDB dependency graph as follows. First, we perform a *clique reduction* which reduces each clique to one node, and the graph is then reduced to a DAG (directed acyclic graph). A node so reduced is circled if the original clique contains a circled node. An IDB is **well-defined** if each source node (the node with in-degree = 0) of the reduced graph is circled (otherwise, there is no proper exit for the recursion). We assign each source node to

level 1 and each other node to an incremental level based on the partial order obtained by a topological sort.

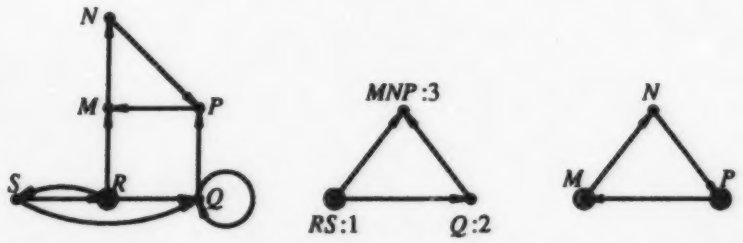
An IDB usually contains several deduction levels. However, query processing acrossing levels does not pose great challenges because it is nonrecursive. A lower level predicate can be viewed as an EDB predicate in the processing of the predicates at the current level. Therefore, our analysis of mutual recursions focuses on one clique. To isolate a clique from others, we (i) treat an IDB predicate as an EDB predicate if its deduction level is lower than the one being considered, (ii) treat a rule whose body consists solely of the predicates each with its level lower than the level of the head predicate as an **exit rule** (which corresponds to a circled node in the IDB dependency graph), and (iii) toss away the rules not directly defining the predicates in the current clique.

**Example 1**. Suppose we have the following rule set in the IDB.

$(r_1)$ $R :- E_1, S, E'_1$. $(r_2)$ $S :- E_2, R, E'_2$. $(r_3)$ $M :- P, E'_3$.
$(r_4)$ $P :- E_4, N, E'_4$. $(r_5)$ $P :- E_5, Q$. $(r_6)$ $N :- E_6, M, E'_6$.
$(r_7)$ $M :- E_7, R, E'_7$. $(r_8)$ $Q :- R, E'_8$. $(r_9)$ $R :- E$.
$(r_{10})$ $Q :- S, Q$.

Its IDB dependency graph is Fig. 1a. According to Algorithm 1, all the $E$'s (EDB predicates) are at level 0, the IDB predicates $R$ and $S$ are at level 1, $Q$ at level 2 and $M$, $N$ and $P$ are at level 3. This can be easily obtained by the clique reduction on the graph. The clique reduction reduces the graph to a DAG (Fig. 1b). Clearly, the clique $RS$ is at level 1, $Q$ at level 2 and $MNP$ at level 3. The IDB is well-defined because the source clique $RS$ is a circled node (since $R$ is circled in Fig. 1a).

Each clique can be analyzed independently of other cliques. For example, the rules $(r_3)$ — $(r_7)$ are the rules for clique $MNP$ (level 3), in which $(r_5)$ and $(r_7)$ are treated as exit rules. The IDB dependency graph of the isolated clique $MNP$ is Fig. 1c. □




a) An IDB dependency graph b) clique reduction c) the clique MNP

**Figure 1.** An IDB dependency graph and the clique reduction.

**Definitions.** A recursive rule is **linear recursive** if its body contains exactly one predicate defined at the same deduction level as that of the head predicate, and the predicate is the same as the head predicate. A set of rules forms a **single/multiple linear recursion** if it consists of single/multiple linear recursive rule(s) and one or more exit rules.

**Definitions.** A recursive rule is **linear mutual recursive** if its body contains exact one predicate defined at the same deduction level as that of the head predicate, and the predicate is *not* the same as the head predicate. A set of rules forms a **linear mutual recursion** if it consists of linear mutual recursive rules, possibly some linear recursive rules, and one or more exit rules.

Clearly, the rule set of Ex. 1 is a linear mutual recursion. Our discussion of the compilation and evaluation of linear mutual recursions focuses on one clique because a lower level IDB predicate can be treated as an EDB predicate in the level-crossing deduction. The IDB dependency graph of a clique is a strongly connected component in which each edge corresponds to a recursive rule and each circled node represents a set of exit rules defining the node. Obviously, there must be at least one circle node in a well-defined clique.

Moreover, our study focuses on the *chain-rule recursions* in which all the recursive rules are *chain rules* [4]. As a notational shorthand, we omit the variables inside the predicates of a chain rule (as shown in Ex. 1) because they are implied by the predicate positions. It is a challenging task to compile and efficiently evaluate *non-chain-rule mutual recursions* (i.e., those containing nonchain rules). We briefly address this issue in Section 5 and leave a detailed study to the future.

## 3. Compilation of Linear Mutual Recursions

As discussed in Section 2, the IDB dependency graph of a linear mutual recursion contains cycles. A node shared by more than one cycle is called a **pivot**. In this section, we classify a linear mutual recursion into the following five classes according to the different kinds of cycles in its IDB dependency graph:

(1) *single cycle*: the graph consists of one cycle only;

(2) *multiple cycle, single pivot*: the graph consists of multiple cycles with only one pivot;

(3) *multiple cycle, ring pivots*: the graph consists of multiple cycles and the pivots form a ring along one cycle;

(4) *chain cycles*: the graph consists of a sequence of cycles which are connected as a chain, that is, each cycle is connected to two neighboring cycles except that at each end of the chain; and

(5) *complex cycle connections*: the graph cotains complex cycle connections beyond the above four classes.

Each of the following subsections is dedicated to the compilation of one of the above classes.

The compilation derives a set of compiled recursive rule(s) and exit rule(s). A **compiled recursive rule** is a recursive rule whose body consists of one IDB predicate and all the remaining predicates are EDB predicates, while a **compiled exit rule** is a rule whose body consists of only EDB predicates. The compilation of an IDB predicate can be viewed as the derivation of all the possible traversal paths from the IDB predicate in the IDB dependency graph. Since an arrow in the IDB dependency graph indicates the direction of implication from a predicate in the body to the head predicate, the derivation of compiled rules for a head predicate should traverse in the *reverse* direction of an arrow.

### 3.1. Class I: single cycle

We first examine a typical linear mutual recursion of Class I.

**Example 2.** The IDB dependency graph (Fig. 2) of the following rule set consists of a single cycle.

$(a_1)$ $R_1 :- A_1, R_2, A'_1.$ $(a_2)$ $R_2 :- A_2, R_3, A'_2.$ ...

$(a_{n-1})$ $R_{n-1} :- A_{n-1}, R_n, A'_{n-1}.$ $(a_n)$ $R_n :- A_n, R_1, A'_n.$ $(e_1)$ $R_1 :- E_1.$

$(e_3)$ $R_3 :- E_3.$ $(e_n)$ $R_n :- E_n.$

The compilation is performed as follows. First, the set of compiled recursive rules of $R_1$ is the set of traversal paths starting at $R_1$ and going back to $R_1$ (without going through $R_1$ itself). Since there is only one such traversal path, "$a_1, a_2, ..., a_n$", in the IDB dependency graph, there is one recursive rule (*3.1a*) which is obtained by performing a sequence of unifications of the recursive rules, that is, unifying $R_i$ in the body with $R_i$ in the head for $i = 2, ..., n-1$. Secondly, the compiled exit rule of $R_1$ is a traversal from $R_1$ to a circled node without going through a cycle containing $R_1$. Since there are three circled nodes at $R_1$, $R_3$ and $R_n$, there are three such paths "$e_1$", "$a_1, a_2, e_3$", and "$a_1, a_2, ..., a_{n-1}, e_n$". Therefore, the exit rule set of $R_1$ should contain three exit rules

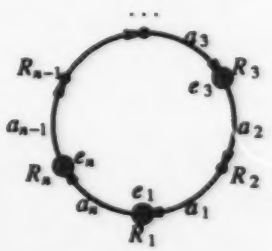



**Figure 2.** An IDB dependency graph of Class I (single cycle).

which can be unioned to one in the form of (*3.1b*), where the notion "$R :- E_1; E_2; E_3$." represents a set of exit rules, "$R :- E_1$.", "$R :- E_2$.", and "$R :- E_3$.".

(*3.1a*) $R_1 :- A_1, A_2, ..., A_n, R_1, A'_n, ..., A'_2, A'_1$.

(*3.1b*) $R_1 :- E_1;\ A_1, A_2, E_3, A'_2, A'_1\ ;\ A_1, ..., A_{n-1}, E_n, A'_{n-1}, ..., A'_1$.

Obviously, the remaining IDB predicates in the clique can be derived from $R_1$ as shown below. Notice that the derivation of each IDB predicate is a traversal from the corresponding IDB predicate to $R_1$.

(*3.1c*) $R_n :- A_n, R_1, A'_n$.

. . . . . .

(*3.1d*) $R_2 :- A_2, ..., A_n, R_1, A'_n, ..., A'_2$. □

**Theorem 2.** *A linear mutual recursion of Class I is compilable to a single linear recursion.*

Proof sketch.

Since Ex. 2 represents a general Class I recursion, its compilation in Ex. 2 proves the theorem. □

We examine the condition in which the recursion can be processed by transitive closure operations. From the compiled recursive rule (*3.1a*), it is clear that if (i) $A_1$, ..., and $A_n$, or (ii) $A'_1$, ..., and $A'_n$, are missing in the recursive rules of the rule set of Ex. 2, the recursion is a one-sided recursion [10], which can be compiled to the form of a transitive closure, (i) $E, A'^*$, or (ii) $A^*, E$, respectively, where $A = A_1, A_2, ..., A_n$; $A' = A'_n, ..., A'_2, A'_1$; $E = E_1 \cup (A_1, A_2, E_3) \cup (A_1, ..., A_{n-1}, E_n)$; and $E' = E_1 \cup (E_3, A'_2, A'_1) \cup (E_n, A'_{n-1}, ..., A'_1)$. Thus we have,

**Corollary.** *If the nonrecursive predicates reside only at one side of the compiled recursive rule in a Class I recursion, the recursion can be processed by a transitive closure operation.*

### 3.2. Class II: multiple cycle, single pivot

We then examine a typical linear mutual recursion of Class II.

**Example 3.** The IDB dependency graph (Fig. 3) of the following rule set consists of multiple cycles with one pivot point $R_1$.

($a_1$) $R_1 :- A_1, R_1, A'_1$.
...
...
($e_3$) $R_3 :- E_3$.

($a_2$) $R_1 :- A_2, R_2, A'_2$.
($a_n$) $R_1 :- A_n, R_n, A'_n$.
($b_n$) $R_n :- B_n, R_1, B'_n$.
($e_n$) $R_n :- E_n$.

($a_3$) $R_1 :- A_3, R_3, A'_3$.
($b_2$) $R_2 :- B_2, R_1, B'_2$.
($e_1$) $R_1 :- E_1$.

The compiled recursive rule set of $R_1$ is the set of traversals from $R_1$ to $R_1$ without going through $R_1$, that is, "$a_1$", "$a_2, b_2$", "$a_3, b_3$", ..., and "$a_n, b_n$". Similarly, the compiled exit rule set of $R_1$ is the set of traversals from $R_1$ to the circled nodes $e_1$, $e_3$ and

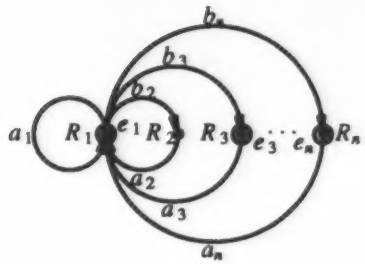



**Figure 3**. An IDB dependency graph of Class II (multiple cycles, single pivot).

$e_n$ without going through $R_1$, that is, "$e_1$", "$a_3, e_3$", and "$a_n, e_n$". Thus the compiled recursion for $R_1$ is a multiple linear recursion which consists of $n$ linear recursive rules and one unioned exit rule as below.

(*3.2a*) $R_1 :- A_1, R_1, A'_1$.

(*3.2b*) $R_1 :- A_2, B_2, R_1, B'_2, A'_2$.

. . . . . .

(*3.2c*) $R_1 :- A_n, B_n, R_1, B'_n, A'_n$.

(*3.2d*) $R_1 :- E_1;\ A_3, E_3, A'_3;\ A_n, E_n, A'_n$.

For notational convenience, the set of compiled linear recursive rules, (*3.2a*) to (*3.2c*), can be abbreviated to one rule (*3.2a'*) using the matrix notation.

(*3.2a'*) $R_1 :- \begin{bmatrix} A_1 \\ A_2, B_2 \\ \vdots \\ A_n, B_n \end{bmatrix}, R_1, \begin{bmatrix} A'_1 \\ B'_2, A'_2 \\ \vdots \\ B'_n, A'_n \end{bmatrix}$.

Obviously, the remaining IDB predicates in the clique can be derived from $R_1$ as below.

(*3.2e*) $R_2 :- B_2, R_1, B'_2$.

. . . . . .

(*3.2f*) $R_n :- B_n, R_1, B'_n$. □

In general, we have

**Theorem 3**. *A linear mutual recursion of Class II is compilable to a multiple linear recursion.*

Proof sketch.

Based on the reasoning in Ex. 3. □

We examine the condition in which a Class II recursion can be processed by transitive closure operations. Clearly, if (i) $A_1$ or $A'_1$, and (ii) $A_i$ and $B_i$, or $B'_i$ and $A'_i$, for $2 \le i \le n$, are missing in the recursive rules of Ex. 3, the recursion can be compiled to the form of one or a set of transitive closures [5]. For example, if (i) $A'_1$, (ii) $A'_n$, and (iii) $A_i$ and $B_i$, for $2 \le i \le n-1$, are missing, the recursion can be compiled to (*3.2g*).

(*3.2g*) $R_1 = A^*, E, B^*$.

where $A = A_1 \cup A_n$; $E = E_1 \cup (E_3, A'_3) \cup (A_n, E_n)$; and $B = (B'_2, A'_2) \cup \cdots \cup (B'_{n-1}, A'_{n-1})$.

**Corollary**. *If the nonrecursive predicates reside at only one side in each cycle of the IDB dependency graph of a Class II recursion, it can be processed by one or two transitive closure operations.*

### 3.3. Class III: multiple cycle, ring pivots

We examine the compilation of a typical linear mutual recursion of Class III.

**Example 4**. The IDB dependency graph of the following rule set is Fig. 4 whose $n$ pivots $R_1, R_2, \ldots$, and $R_n$ form a ring.

$(a_1)$ $R_1 :- A_1, R_2, A'_1$.
$(a_2)$ $R_2 :- A_2, R_3, A'_2$.
$\cdots$
$(a_n)$ $R_n :- A_n, R_1, A'_n$.
$(b_1)$ $R_1 :- B_1, R_1, B'_1$.
$(b_2)$ $R_2 :- B_2, R_2, B'_2$.
$\cdots$
$(b_n)$ $R_n :- B_n, R_n, B'_n$.
$(e_1)$ $R_1 :- E_1$.
$(e_3)$ $R_3 :- E_3$.
$(e_n)$ $R_n :- E_n$.

The compiled recursive rule set of $R_1$ is derived by all the possible traversals from $R_1$ to $R_1$ (without going through $R_1$), which consists of (i) a simple cycle "$b_1$", and (ii) a path "$a_1, (b_2), a_2, \ldots, (b_n), a_n$", where the portion in a pair of parentheses such as $(b_2)$ indicates that the portion $b_2$ is traversed 0 or more times. Such a traversal will symmetrically derive a set of nonrecursive predicates at both sides of $R_1$ in the body, that

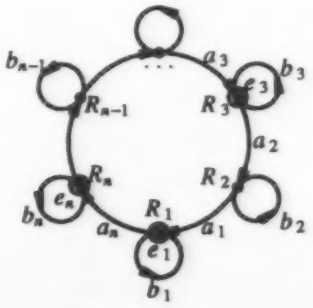



**Figure 4**. An IDB dependency graph of Class III (multiple cycle, ring pivots).

is, for rule $b_2$, there will be $B_2^{i_2}$ (and $B'^{i_2}_2$) (for $i_2 \geq 0$) at the left (and right) side of $R_1$ respectively. The compiled exit rule set of $R_1$ is derived by traversing the path from $R_1$ to a set of circled nodes, that is, (i) "$e_1$", (ii) "$a_1, (b_2), a_2, (b_3), e_3$", and (iii) "$a_1, (b_2), a_2, (b_3), ..., (b_n), e_n$". Thus we have

(*3.3a*) $R_1 :- B_1, R_1, B'_1$.

(*3.3b*) $R_1 :- A_1, B_2^{i_2}, A_2, B_3^{i_3}, ..., B_n^{i_n}, A_n, R_1, A'_n, B'^{i_n}_n, ..., B'^{i_3}_3, A'_2, B'^{i_2}_2, A'_1$.

(*3.3c*) $R_1 :- E_1$.

(*3.3d*) $R_1 :- A_1, B_2^{i_2}, A_2, B_3^{i_3}, E_3, B'^{i_3}_3, A'_2, B'^{i_2}_2, A'_1$.

(*3.3e*) $R_1 :- A_1, B_2^{i_2}, A_2, B_3^{i_3}, ..., B_n^{i_n}, E_n, B'^{i_n}_n, ..., B'^{i_3}_3, A'_2, B'^{i_2}_2, A'_1$.

where $i_j \geq 0$ for $j$ from 2 to $n$. Notice that the rules (*3.3b*), (*3.3d*), and (*3.3e*) contain some predicates in exponential forms.

Similar to Class II, the set of linear recursive rules, (*3.3a*) and (*3.3b*), can be abbreviated to one rule (*3.3a'*) using the matrix notation.

(*3.3a'*) $R_1 :- \begin{bmatrix} B_1 \\ A_1, B_2^{i_2}, A_2, B_3^{i_3}, ..., B_n^{i_n}, A_n \end{bmatrix}, R_1, \begin{bmatrix} B'_1 \\ A'_n, B'^{i_n}_n, ..., B'^{i_3}_3, A'_2, B'^{i_2}_2, A'_1 \end{bmatrix}$.

The remaining IDB predicates in the clique can be derived from $R_1$ and written in exponential forms of some predicates as below.

(*3.3e*) $R_n :- B_n^{i_n}, A_n, R_1, A'_n, B'^{i_n}_n$.

. . . . . .

(*3.3f*) $R_2 :- B_2^{i_2}, A_2, B_3^{i_3}, ..., B_n^{i_n}, A_n, R_1, A'_n, B'^{i_n}_n, ..., B'^{i_3}_3, A2', B'^{i_2}_2$.

where $i_j \geq 0$ for $j$ from 2 to $n$. □

We define a rule whose body contains predicates represented in exponential forms as below.

**Definitions**. A rule whose body contains some EDB predicate(s) in an exponential form, representing zero or more appearances of the EDB predicate(s) in the rule, is called a **rule in exponential form**. A linear recursive rule in exponential form is a **pseudo-linear recursive rule**. An exit rule in exponential form is a **pseudo-exit rule**. A recursion is a **pseudo-linear recursion** if it consists of one or more pseudo-linear recursive rules, zero or more linear recursive rules, and one or more (pseudo-) exit rules.

Clearly, the rule set of (*3.3a*) to (*3.3e*) is a pseudo-linear recursion in which (*3.3a*) is a linear recursive rule, (*3.3b*) is a pseudo-linear recursive rule, and (*3.3c*) to (*3.3e*) are (pseudo)-exit rules.

There could be more complex forms of recursions than Ex. 4 in some Class III recursion, in which there may exist multiple "small" cycles at each pivot. For example, instead of one cycle, "$R_2 :- B_2, R_2, B'_2$.", we may have $m$ "small" cycles on the pivot $R_2$ ($m$ linear recursive rules) as below,

$R_2 :- B_{21}, R_2, B'_{21}.$

. . . . . .

$R_2 :- B_{2m}, R_2, B'_{2m}.$

In this case, the compiled pseudo-linear recursive rule set should be the same as the original one except the substitution of $B_2$ and $B'_2$ by a matrix as below,

$$B_2 = \begin{bmatrix} B_{21} \\ \vdots \\ B_{2m} \end{bmatrix}, \quad \textit{and} \quad B'_2 = \begin{bmatrix} B'_{21} \\ \vdots \\ B'_{2m} \end{bmatrix}.$$

In general, we have

**Theorem 4.** *A linear mutual recursion of Class III is compilable to a pseudo-linear recursion.*

Proof sketch.

Based on the reasoning in Ex. 4. □

We examine the condition in which Class III recursions can be processed by a set of transitive closure operations. Similar to the previous classes, if (i) $B_1$ or $B'_1$, and (ii) $A_i$ and $B_{i+1}$, or $A'_i$ and $B'_{i+1}$, for $1 \le i \le n-1$, are missing in the rule set of Ex. 4, the recursion can be processed by a set of transitive closure operations [5].

**Corollary.** *If the nonrecursive predicates reside at only one side in each recursive rule obtained by resolving each cycle in the IDB dependency graph of a Class III recursion, the recursion can be processed by a set of transitive closure operations.*

### 3.4. Class IV: chain cycles

We examine a typical linear mutual recursion of Class IV.

**Example 5.** The IDB dependency graph of the following rule set is in Fig. 5, which consists of a chain of connected cycles.

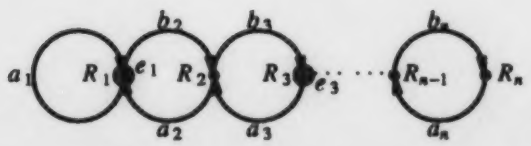



**Figure 5.** An IDB dependency graph of Class IV (chain cycles).

$(a_1)$ $R_1 :- A_1, R_1, A'_1.$ $(a_2)$ $R_1 :- A_2, R_2, A'_2.$ $(b_2)$ $R_2 :- B_2, R_1, B'_2.$

$(a_3)$ $R_2 :- A_3, R_3, A'_3.$ $(b_3)$ $R_3 :- B_3, R_2, B'_3.$ ...

$(a_n)$ $R_{n-1} :- A_n, R_n, A'_n.$ $(b_n)$ $R_n :- B_n, R_{n-1}, B'_n.$ $(e_1)$ $R_1 :- E_1.$

$(e_3)$ $R_3 :- E_3.$

The compiled recursive rule set of $R_1$ is derived by all the possible traversals from $R_1$ to $R_1$ without going through $R_1$, which consists of (i) a simple cycle "$a_1$", and (ii) a path "$a_2, (a_3, (\ldots (a_n, b_n) \ldots), b_3), b_2$", where the portion in a pair of parentheses is traversed zero or more times, which symmetrically derives a set of nonrecursive predicates at both sides of $R_1$. Similarly, the compiled exit rule set of $R_1$ is derived by all the possible traversals from $R_1$ to a set of circled nodes without going through $R_1$, that is, (i) "$e_1$", and (ii) all the paths from $R_1$ to $e_3$, i.e., starting at $R_1$, going through "$a_2, a_3$" (reaching $R_3$), and then traversing all the paths from $R_3$ to $R_3$ without going through $R_1$, which itself is a subcase of chain cycles, i.e., starting at $R_3$, going through zero or more times of (i) a cycle "$b_3, a_3$" (without going through $R_1$), or (ii) a set of cycles "$b_4, (\ldots b_n, a_n) \ldots), a_4$", and finally reaches $e_3$. Therefore, we have

$(3.4a)$ $R_1 :- A_1, R_1, A'_1.$

$(3.4b)$ $R_1 :- A_2, (A_3, \ldots (A_n, B_n)^{i_n}, \ldots, B_3)^{i_3}, B_2, R_1, B'_2, (B'_3, \ldots (B'_n, A'_n)^{i_n}, \ldots, A'_3)^{i_3}, A'_2.$

$(3.4c)$ $R_1 :- E_1.$

$(3.4d)$ $R_1 :- A_2, A_3, R'_3, A'_3, A'_2.$

$(3.4e)$ $R'_3 :- B_3, A_3, R'_3, A'_3, B'_3.$

$(3.4f)$ $R'_3 :- B_4, (B_5, \ldots (B_n, A_n)^{i_n}, \ldots, A_5)^{i_5}, A_4, R'_3, A'_4, (A'_5, \ldots (A'_n, B'_n)^{i_n}, \ldots, B'_5)^{i_5}, B'_4.$

$(3.4g)$ $R'_3 :- E_3.$

where $i_j \geq 0$ for $j$ from 3 to $n$. Notice that the exponential forms in the body of each psuedo-recursive or pseudo-exit rule are symmetric to the recursive predicate in the body.

Clearly, the compiled recursion is a pseudo-linear recursion in which the compiled recursive rule set, $(3.4a)$ and $(3.4b)$, can be represented in a matrix form as $(3.4a')$. A similar matrix form can also be given for $(3.4e)$ and $(3.4f)$. □

$$(3.4a') \quad R_1 :- \begin{bmatrix} A_1 \\ A_2, (A_3, \ldots, (A_n, B_n)^{i_n}, \ldots, B_3)^{i_3}, B_2 \end{bmatrix}, R_1, \begin{bmatrix} A'_1 \\ B'_2, (B'_3, \ldots, (B'_n, A'_n)^{i_n}, \ldots, A'_3)^{i_3}, A'_2 \end{bmatrix}.$$

In general, we have

**Theorem 5.** *A linear mutual recursion of Class IV is compilable to a pseudo-linear recursion.*

Proof sketch.

Based on the reasoning in Ex. 5. □

We examine the condition in which Class IV recursions can be processed by a set of transitive closure operations. Similar to the previous classes, if (i) $A_1$ or $A'_1$, and (ii) $A_i$

and $B_i$, or $A'_i$ and $B'_i$, for $2 \le i \le n$, are missing in the rule set of Ex. 5, the recursion can be processed by a set of transitive closure operations [5].

**Corollary**. *If the nonrecursive predicates reside at only one side in each recursive rule obtained by resolving each cycle in the IDB dependency graph of a Class IV recursion, the recursion can be processed by a set of transitive closure operations.*

### 3.5. Class V: complex multiple cycles

A Class V recursion whose IDB dependency graph consists of complex connected cycles can be viewed as a combination of previously studied cases. For example, the recursion may consist of a chain of cycles, along each cycle there may be a ring of pivots, and each pivot may be shared by a set of "small" cycles. The compilation is also a combination of the previously studied cases: (i) a pivot shared by a set of "small" cycles can be compiled to a set of multiple linear recursive rules (Class II); (ii) a set of ring pivots can be compiled to a pseudo-linear recursive rule (Class III); and (iii) the set of chain cycles can be compiled to another set of pseudo-linear recursive rules (Class IV).

There could be many cycles in a complex inter-connected graph. The task of the compilation is to find all the possible cycle connections to guarantee that all the deduction paths are searched without overlapping. We examine one example.

**Example 6**. Suppose we have the following rule set with IDB dependency graph Fig. 6.

| | | |
|---|---|---|
| $(a_2)$ $R_1 :- A_2, R_2, A'_2.$ | $(b_2)$ $R_2 :- B_2, R_1, B'_2.$ | $(a_3)$ $R_2 :- A_3, R_3, A'_3.$ |
| $(b_3)$ $R_3 :- B_3, R_2, B'_3.$ | $(a_4)$ $R_3 :- A_4, R_4, A'_4.$ | $(b_4)$ $R_4 :- B_4, R_3, B'_4.$ |
| $(a_1)$ $R_1 :- A_1, R_4, A'_1.$ | $(b_1)$ $R_4 :- B_1, R_1, B'_1.$ | $(e_1)$ $R_1 :- E_1.$ |

Starting at $R_1$, there are four possible nonoverlapping cycle connections:

1. cycle $a_2b_2$ with 3 chain cycles on its top: $a_3b_3$ , $a_4b_4$, and $b_1a_1$;

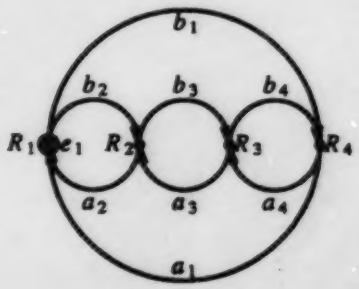



**Figure 6.** An IDB dependency graph of Class V.

2. cycle $a_1b_1$ with 3 chain cycles on its top: $b_4a_4$, $b_3a_3$, and $b_2a_2$;
3. cycle $a_2a_3a_4b_1$ on which there are 4 ring cyles: $b_2a_2$ at $R_2$, $b_3a_3$ at $R_3$, $b_4a_4$ at $R_4$, and $a_1b_1$ at $R_1$; and
4. cycle $a_1b_4b_3b_2$ on which there are 4 ring cycles: $b_1a_1$ at $R_1$, $a_4b_4$ at $R_4$, $a_3b_3$ at $R_3$, and $a_2b_2$ at $R_2$.

Notice that a cycle may have two pivots along the same cycle, such as a cycle $a_2b_2$ with two pivots $R_1$ and $R_2$ along the cycle $a_2a_3a_4b_1$. The two pivots count as one in the computation of ring cycles because the traversals along them generate the same set of paths. There are four possible traversal paths in the derivation of the compiled recursive rule set:

1. $a_2, (a_3, (a_4, (b_1, a_1), b_4), b_3), b_2$,
2. $a_1, (b_4, (b_3, (b_2, a_2), a_3), a_4), b_1$,
3. $a_2, (b_2, a_2), a_3, (b_3, a_3), a_4, (b_4, a_4), b_1, (a_1, b_1)$, and
4. $a_1, (b_1, a_1), b_4, (a_4, b_4), b_3, (a_3, b_3), b_2, (a_2, b_2)$.

Therefore, the recursion can be compiled to the following set of pseudo-linear recursions for $R_1$:

(3.5a) $R_1 :- A_2, (A_3, (A_4, (B_1, A_1)^{i_1}, B_4)^{i_4}, B_3)^{i_3}, B_2, R_1,$
$B'_2, (B'_3, (B'_4, (A'_1, B'_1)^{i_1}, A'_4)^{i_4}, A'_3)^{i_3}, A'_2.$

(3.5b) $R_1 :- A_1, (B_4, (B_3, (B_2, A_2)^{i_2}, A_3)^{i_3}, A_4)^{i_4}, B_1, R_1,$
$B'_1, (A'_4, (A'_3, (A'_2, B'_2)^{i_2}, B'_3)^{i_3}, B'_4)^{i_4}, A'_1.$

(3.5c) $R_1 :- A_2, (B_2, A_2)^{i_2}, A_3, (B_3, A_3)^{i_3}, A_4, (B_4, A_4)^{i_4}, B_1, (A_1, B_1)^{i_1}, R_1,$
$(B'_1, A'_1)^{i_1}, B'_1, (A'_4, B'_4)^{i_4}, A'_4, (A'_3, B'_3)^{i_3}, A'_3, (A'_2, B'_2)^{i_2}, A'_2.$

(3.5d) $R_1 :- A_1, (B_1, A_1)^{i_1}, B_4, (A_4, B_4)^{i_4}, B_3, (A_3, B_3)^{i_3}, B_2, (A_2, B_2)^{i_2}, R_1,$
$(B'_2, A'_2)^{i_2}, B'_2, (B'_3, A'_3)^{i_3}, B'_3, (B'_4, A'_4)^{i_4}, B'_4, (A'_1, B'_1)^{i_1}, A'_1.$

(3.5e) $R_1 :- E_1.$

The compiled recursive rule set can be written into matrix form as well. The remaining IDB predicates can be easily derived from $R_1$. □

In general, we have

**Theorem 6.** *A linear mutual recursion of Class V is compilable to a pseudo-linear recursion.*

Proof sketch.

Based on the analysis in Ex. 6, the IDB dependency graph of a complex linear mutual recursion in Class V can be viewed as a set of subgraphs, each of which is a combination of cycles of the previous four classes. Since each of them is compilable to a (pseudo-) linear recursion, so are their combinations. Thus we have the theorem. □

The example also shows that a small set of linear mutual recursive rules may generate a large set of compiled (pseudo-) linear recursive rules. In general, we have

**Theorem 7.** *The size of the compiled pseudo-linear recursion of a complex linear mutual recursion may grow exponentially in the worst case.*

Proof sktech.

Suppose the IDB dependency graph of an IDB is a fully connected directed graph of its $n$ IDB predicates $R_1, ..., R_n$. Since there is one edge from each node (IDB predicate) to the other $n-1$ nodes, the graph has $n \times (n-1)$ edges, each corresponding to one mutual recursive rule in the IDB. The compilation of $R_1$ requires the traversal of all the distinct paths from $R_1$ to $R_1$ without going through $R_1$. Let $P(n,r)$ denote the number of $r$-permutations of $n$ distinct elements without repetitions and $C(n,r)$ denote the number of $r$-combinations of $n$ distinct elements without repetitions. There are $P(n-1,r)$ such paths by starting at $R_1$, going through $r$ nodes (out of the $n-1$ nodes), and then returning to $R_1$. Therefore, the total number of distinct paths is

$$\sum_{r=1}^{n-1} P(n-1,r) \geq \sum_{r=1}^{n-1} C(n-1,r) = 2^{n-1} - 1.$$

where $n > 1$. Since each path will generate one (pseudo-) linear recursive rule, the size of this compiled recursion grows exponentially when $n$ is large. □

Obviously, it is elegant to compile and process a linear mutual recursion in Class I or II since the compiled recursion is simple and in small size. However, when there are dense connections among recursive predicates, the size of the compiled recursion may grow exponentially.

## 4. Efficient Evaluation of Linear Mutual Recursions

In this section, we show that our compilation benefits the evaluation of linear mutual recursions.

First, compilation distinguishes the linear mutual recursions which can be processed by transitive closure operations. As a summary of the corollaries of Section 3, we have

**Corollary.** *If each recursive rule obtained by resolving each cycle in the IDB dependency graph of a linear mutual recursion is a one-sided rule [5, 10], the recursion can be evaluated by a set of transitive closure operations.*

Moreover, many nested transitive closure operations can be simplified to the operations containing no nested transitive closures, e.g., $(A^+B^+)^+ \equiv A(A \cup B)^*B$ [5].

Secondly, a linear mutual recursion can be compiled to one or a set of linear or pseudo-linear recursions. Therefore, its evaluation is essentially the evaluation of a set of (pseudo-) linear recursions. There have been many intersting techniques developed for the efficient evaluation of linear recursions, such as the magic sets method [1], the counting method [1], the Henschen-Naqvi algorithm [8], the query and sub-query approach [13], etc. What we need is their extension(s) to the evaluation of compiled (pseudo-)linear recursions.

Since the magic sets method is a general bottom-up evaluation method which works on both cyclic and acyclic data, our discussion focuses on the application of the magic

sets method. Assuming that the query is in the form of "$?-x=c, R_1(x,y)$.", we examine the evaluation of $R_1$ in each class. The other IDB predicates in the recursion can be derived from $R_1$ straightforwardly.

We first examine a typical Class I recursion shown in Ex. 2. Since the recursion is compiled to a single linear recursion, it can be evaluated by a typical linear recursive query evaluation method. The compilation benefits the processing since (i) it transforms multiple mutual recursive rules into one linear recursive rule, (ii) preprocessing can be performed to merge (i.e., join) the predicates at each side of the recursive predicate in the body, such as $A = A_1, A_2, ..., A_n$ and $A' = A'_n, ..., A'_2, A'_1$, which reduces the number of joins to be performed in the query processing, and (iii) if it is evaluated by the magic set method, the magic set is simply a query-relevant transitive closure of $A$, that is, $\sigma_{x=c}A^*$.

We then examine a typical Class II recursion shown in Ex. 3. Since the recursion is compiled to a multiple linear recursion, it can be evaluated by a typical multiple linear recursion evaluation method, such as the side-relation unioned magic set method [7]. We briefly illustrate the method using the following example.

**Example 7.** Side-relation unioned magic set evaluation of the query, "$?-R_1(c,y)$.", inquired on the Class II recursion with the following compiled rules.

$$(4.1a)\quad R_1 :- \begin{bmatrix} A_1 \\ A_2, B_2 \\ \vdots \\ A_n, B_n \end{bmatrix}, R_1, \begin{bmatrix} A'_1 \\ B'_2, A'_2 \\ \vdots \\ B'_n, A'_n \end{bmatrix}.$$

$$(4.1b)\quad R_1 :- E.$$

The evaluation is performed in the following three steps.

1. Perform *side-information associated union* of the relations at each side. Such a union is denoted by "$\cup$" which indicates that each tuple of the unioned relation carries the information about which original relation the tuple is from. We have

   $A = A_1 \cup (A_2, B_2) \cup \cdots \cup (A_n, B_n)$.
   $B = A'_1 \cup (B'_2, A'_2) \cup \cdots \cup (B'_n, A'_n)$.

2. Derive (i) the magic set, $Magic(x)$, the transitive closure of $A$ relevant to the query probe $c$, and (ii) $A'$, the query relevant portion of $A$, where "$A'(x,y) :- A(x,y), Magic(x)$."

3. Perform side-relation matched semi-naive evaluation. That is, starting at $E'$ where "$E'(x,y) = Magic(x), E(x,y)$", perform a *side-matched* semi-naive evaluation on $A'$ and $B$, which requires the match of the original side-relation pairs between a driving tuple and a driven tuple at each side.

The final answer to the query is the set obtained by performing selection on the result closure using the query constant. □

The method saves interleaved accessing of multiple relations by unioning the relations at one side and thus improves the performance. A detailed analysis of the technique is in [7].

Then we examine the efficient evaluation of the compiled pseudo-linear recursions.

**Example 8.** The magic set evaluation of the query, "$?-R_1(c, y)$.", inquired on a Class III recursion with the following compiled rules.

(4.1a') $R_1 :- \left[ \begin{matrix} B_1 \\ A_1, B_2^{i_2}, A_2, B_3^{i_3}, \ldots, B_n^{i_n}, A_n \end{matrix} \right], R_1, \left[ \begin{matrix} B'_1 \\ A'_n, B'^{i_n}_n, \ldots, B'^{i_3}_3, A'_2, B'^{i_2}_2, A'_1 \end{matrix} \right].$

(4.1b) $R_1 :- E.$

The evaluation is performed in the following two steps.

1. Derive (i) the magic set, $Magic(x)$, which is the query-relevant transitive closure of
   $$B_1 \cup (A_1, B_2^*, A_2, B_3^*, \ldots, B_n^*, A_n),$$
   and (ii) the query-relevant portion of the left side relations; and
2. Perform side-relation matched semi-naive evaluation. That is, starting at $E'$ where "$E'(x, y) = Magic(x), E(x, y)$", perform the semi-naive evaluation on the two sides of $R_1$. At each iteration, first perform side-relation matched join on the side-information unioned relations $B_1 \cup A_n$ and $B'_1 \cup A'_n$. Then for those obtained by matching $A_n$ and $A'_n$, evaluation proceeds by (i) performing semi-naive evaluation on $B_n$ and $B'_n$, (ii) performing join on $A_{n-1}$ and $A'_{n-1}$, (iii) performing semi-naive evalaution on $B_{n-1}$ and $B'_{n-1}$, and so on, until it reaches the final one, i.e., performing join on $A_1$ and $A'_1$. The tuples so obtained are (side-information associated) unioned with those obtained by processing $B_1$ and $B'_1$. Then another iteration of the semi-naive evaluation proceeds. The process terminates when there is no new tuple generated in the evaluation.

The final answer to the query is the set obtained by performing selection on the result closure using the query constant. □

Similarly, we have the method for the evaluation of a Class IV recursion.

**Example 9.** The magic set evaluation of the query, "$?-R_1(c, y)$.", inquired on a Class IV recursion with the following compiled rules.

(4.1a') $R_1 :- \left[ \begin{matrix} A_1 \\ A_2, (A_3, \ldots, (A_n, B_n)^{i_n}, \ldots, B_3)^{i_3}, B_2 \end{matrix} \right], R_1, \left[ \begin{matrix} A'_1 \\ B'_2, (B'_3, \ldots, (B'_n, A'_n)^{i_n}, \ldots, A'_3)^{i_3}, A'_2 \end{matrix} \right].$

(4.1b) $R_1 :- E.$

The evaluation is performed in the following two steps.

1. Derive (i) the magic set, $Magic(x)$, which is the query-relevant transitive closure of
   $$A_1 \cup (A_2, (A_3, \ldots, (A_n, B_n)^*, \ldots, B_3)^*, B_2),$$
   and (ii) the query relevant portion of the left side relations; and

2. Perform side-relation matched semi-naive evaluation. That is, starting at $E'$ where "$E'(x, y) = Magic(x), E(x, y)$", perform the semi-naive evaluation on the two sides of $R_1$. At each iteration, first perform side-relation matched join on the side-information unioned relations $A_1 \cup B_2$ and $A'_1 \cup B'_2$. Then for those obtained by matching $B_2$ and $B'_2$, evaluation proceeds by (i) performing semi-naive evaluation on the two sequences started with $B_3$ and $B'_3$, and (ii) performing join on $A_2$ and $A'_2$, where the semi-naive evaluation of step (i) itself is a nested evaluation in which each evaluation is performed by one evaluation of $B_3$ and $B'_3$, then a semi-naive evaluation on two deeply nested sequences started with $B_4$ and $B'_4$, and then join on $A_3$ and $A'_3$. Finally, the tuples obtained at step (ii) are unioned with those obtained by processing $A_1$ and $A'_1$, with the side-information associated with each tuple. Then another iteration of the semi-naive evalaution proceeds. The process terminates when there is no new tuple generated in the evaluation.

The final answer to the query is the set obtained by performing selection on the result closure using the query constant. □

Clearly, a Class V recursion should be evaluated by a combination of the methods demonstrated in the above examples.

## 5. Discussion

In this section, we discuss the strength and the limitations of our compilation approach.

In comparison with other methods for the evaluation of linear mutual recursions, our compilation has the following advantages. First, it resolves the intermediate predicates along each simple cycle and detects the portions which can be compiled to transitive closure operations. Further optimizations can often be performed on those portions to simply the processing of transitive closures and further improve performance. Secondly, it resolves a set of intermediate relations along each simple cycle to one joined relation, which saves the cost of accessing of several intermediate relations. Thirdly, for multiple simple cycles sharing the same pivots (a Class II recursion), shared processing is explored by side-relation unioned processing, which saves interleaved accessing of multiple relations [7]. Finally, we examine the benefits from compiling the recursions in classes III and IV.

**Example 10**. Suppose a Class IV recursion (similar to Ex. 4) is compiled to the following two rules:

(*5.1a*) $R_1 :- A_1, B_2^{i_2}, A_2, R_1, A'_2, B'^{i_2}_2, A'_1$.
(*5.1b*) $R_1 :- E_1$.

According to our discussion in Section 4, a query such as, "$?- R_1(c, y)$.", can be evaluated by first deriving the magic set based on "$\sigma_{(x=c)}(A_1, B_2^*, A_2)^*$", which can be simplified by the equation [5],

$$(A_1, B_2^*, A_2)^* = (A_1, ((A_2, A_1) \cup B_2)^*, A_2) \cup \phi.$$

Then the semi-naive evaluation is performed on the recursion using the magic set. Our method evaluates the closure path (i.e., $B_2^{i,j}$ and $B'^{i,j}_2$) after the evaluation of $A_2, R_1, A'_2$. One may wonder whether it could be more efficient by first evaluating other paths, for example, first evaluating $A_1$ and $A'_1$. Since the query is to find all the answers for the recursion, the paths left have to be search sooner or later. The search in other orders, e.g., first searching $A_1$ and $A'_1$ without searching the closure path, will increase the database accessing cost because $A_1$ and $A'_1$ have to be fetched again after the search of the closure path, and our search method groups those obtained via different paths as one set before accessing $A_1$ and $A'_1$. □

Our study discloses the exponential growth in the size of a compiled recursion in a densely connected IDB dependency graph in the worst case. One may wonder whether such a result will disfavor our compilation method. Actually, to find *all* the answers in a linear mutual recursion with a densely connected IDB dependency graph, we have to search all the distinct paths no matter what search strategy is used. Our method simply makes explicit the fact of the possible exponential growth of the distinct search paths. Obviously, such an exponential growth challenges all the techniques for finding all the answers in a complex linear mutual recursion.

Our study is confined to chain-rule linear mutual recursions. A linear chain rule is a *well-formed* one- or two- sided recursive rule, in which each variable in the recursive predicate in the body is linked to the variable in the corresponding position in the head predicate via one binary nonrecursive predicate. The compilation technique can be extended to the recursions in which each recursive rule is a well-formed $n$-sided recursive rule for any $n$. This can be seen from the compilation of the following Class I recursion in which each recursive rule is a well-formed $n$-chain rule for $n = 3$.

**Example 11.** The following recursion is a Class I recursion (since its IDB dependency graph is the same as Fig. 2) in which each recursive rule is a well-formed 3-sided recursive rule.

$(a_1)$ $R_1(x_1, x_2, x_3) :- A_1(x_1, x'_1), B_1(x_2, x'_2), C_1(x_3, x'_3), R_2(x'_1, x'_2, x'_3).$

$(a_2)$ $R_2(x_1, x_2, x_3) :- A_2(x_1, x'_1), B_2(x_2, x'_2), C_2(x_3, x'_3), R_3(x'_1, x'_2, x'_3).$

. . . . . .

$(a_n)$ $R_n(x_1, x_2, x_3) :- A_n(x_1, x'_1), B_n(x_2, x'_2), C_n(x_3, x'_3), R_1(x'_1, x'_2, x'_3).$

$(e_1)$ $R_1(x_1, x_2, x_3) :- E_1(x_1, x_2, x_3).$

$(e_3)$ $R_3(x_1, x_2, x_3) :- E_3(x_1, x_2, x_3).$

$(e_n)$ $R_n(x_1, x_2, x_3) :- E_n(x_1, x_2, x_3).$

By the compilation process similar to Ex. 2, that is, performing a sequence of unifications of the recursive rules by unifying $R_i$ in the body with $R_i$ in the head for $i = 2, ..., n-1$, we obtain the compiled recursive rule as below.

(5.2a) $R_1(x_1, x_2, x_3) :- A(x_1, x'_1), B(x_2, x'_2), C(x_3, x'_3), R_1(x'_1, x'_2, x'_3)$.

where

$A(x_1, x'_1) :- A_1(x_1, t_2), A_2(t_2, t_3), ..., A_n(t_n, x'_1)$.
$B(x_1, x'_2) :- B_1(x_1, t_2), B_2(t_2, t_3), ..., B_n(t_n, x'_2)$.
$C(x_1, x'_3) :- C_1(x_1, t_2), C_2(t_2, t_3), ..., C_n(t_n, x'_3)$.

The remaining compilation process is similar to Ex. 2. Obviously, the compilation results in a single linear recursion in which the recursive rule is a well-formed 3-sided recursive rule. □

In general, the variable connections in a mutual recursion can be very complex. It is a challenging task to compile linear mutual recursions with complex variable connections. For a Class I recursion, since the resolution of its sole cycle in the IDB dependency graph transforms the recursion into a single linear recursion, the techniques for the compilation of single linear recursions in complex variable patterns can be applied to compile such a recursion into an n-chain or a bounded recursion [6]. However, it is an open problem to compile other classes of linear mutual recursions with complex variable connections. Also, the compilation of nonlinear mutual recursions is another interesting issue for research.

Interestingly, most mutual recursions found in current logic programming and deductive database programs [9, 11] belong to the linear mutual recursions in Classes I or II or those which can be compiled to a set of transitive closure operations. Other classes of mutual recursions, though challenging in processing, are unlikely to find many applications. Moreover, many applications require to find some answers rather than all the answers for a mutual recursion. Such queries can also be evaluated using the compiled rule set or a portion of it with relaxed termination conditions. In this sense, a full compilation of a mutual recursion may not be necessary for finding some answers instead of all the answers.

## 6. Conclusions

This paper classifies linear mutual recursions and develops some useful techniques for compilation and evaluation of chain-rule linear mutual recursions. It shows that such recursions can be compiled to one or a set of linear recursions or pseudo-linear recursions and evaluated relatively efficiently using the linear recursive query processing techniques and their variations if the compiled recursion is in a reasonable size. However, it also shows that the size of a compiled pseudo-linear recursion for a complex linear mutual recursion may grow exponentially in the worst case.

Our study is confined to chain-rule linear mutual recursions. It is an interesting research issue for efficient compilation and evaluation of non-chain-rule linear mutual recursions and nonlinear mutual recursions in deductive databases.

## References

1. F. Bancilhon, D. Maier, Y. Sagiv and J. D. Ullman, Magic Sets and Other Strange Ways to Implement Logic Programs, *Proceedings of 5th ACM Symposium on Principles of Database Systems*, Cambridge, MA, March 1986, 1–15.
2. F. Bancilhon and R. Ramakrishnan, An Amateur's Introduction to Recursive Query Processing Strategies, *Proceedings of 1986 ACM-SIGMOD Conference on Management of Data*, Washington, DC, May 1986, 16–52.
3. C. Beeri and R. Ramakrishnan, On the Power of Magic, *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, San Diego, CA, March 1987, 269–283.
4. C. Beeri, P. Kanellakis, F. Bancilhon and R. Ramakrishnan, Bounds on the Propagation of Selection into Logic Programs, *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, San Diego, CA, March 1987, 214–226.
5. J. Han and W. Lu, Asynchronous Chain Recursions, *IEEE Transactions on Knoweldge and Data Engineering*, 1(2), 1989, 185–195.
6. J. Han, Compiling General Linear Recursions by Variable Connection Graph Analysis, *Computational Intelligence*, 5(1), 1989, 12–31.
7. J. Han and L. Liu, Processing Multiple Linear Recursions, *Proceedings of 1989 North American Conference on Logic Programming*, Cleveland, OH, October 1989, 816–830.
8. L. J. Henschen and S. Naqvi, On Compiling Queries in Recursive First-Order Databases, *Journal of ACM* , 31(1), 1984, 47–85.
9. S. Naqvi and S. Tsur, *A Logical Data Language for Data and Knowledge Bases*, Computer Science Press, 1989.
10. J. F. Naughton, One-Sided Recursions, *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, San Diego, CA, March 1987, 340–348.
11. L. Sterling and E. Shapiro, *The Art of Prolog*, The MIT Press, 1986.
12. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Vols. 1 & 2*, Computer Science Press, 1989.
13. L. Vieille, Recursive Axioms in Deductive Databases: The Query/Subquery Approach, *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, SC, April 1986, 179–193.
14. N. Zhuang and L. J. Henschen, Compilation of Uniform Linear Recursive Rule Clusters, *unpublished manuscript*, December 1988.

## Centre for Systems Science

---

• Simon Fraser University established the **Centre for Systems Science** as a provincial centre for advanced study. The objective of CSS is twofold: coordinate and combine advanced systems research at SFU, and encourage application of research results within industry in order to further the technological development of British Columbia.

• CSS has assembled three research groups to streamline its efforts. The orientation of these groups is as follows: intelligent systems, microelectronics, and computer and communications systems. Because of shared research interests, this latter group merges with LCCR (**Laboratory for Computer and Communications Research**) for the publication of technical reports.

• CSS's three groups involve over 80 skilled faculty researchers principally from SFU's Schools of Engineering Science and Computing Science, but also draw substantially on Mathematics, Physics, Geography, Philosophy, Linguistics, Psychology, Communication, and Kinesiology.